Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 7, 2016

R. Shakir
BT
A. Shaikh
M. Hines
Google

July 6, 2015

Consistent Modeling of Operational State Data in YANG
draft-openconfig-netmod-opstate-01

Abstract

This document proposes an approach for modeling configuration and operational state data in YANG [RFC6020] that is geared toward network management systems that require capabilities beyond those typically envisioned in a NETCONF-based management system. The document presents the requirements of such systems and proposes a modeling approach to meet these requirements, along with implications and design patterns for modeling operational state in YANG.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 7, 2016.

Table of Contents

1. Introduction

Retrieving the operational state of a network element (NE) is a critical process for a network operator, both because it determines how the network is currently running (for example, how many errors are occurring on a certain link, what is the load of that link); but also because it determines whether the intended configuration applied by a network management system is currently operational. While configuration changes may be relatively infrequent, accessing the state of the network happens significantly more often. Knowing the real-time state of the network is required for a variety of use cases including traffic management, rapid diagnosis and recovery, and enabling tight control loops (implying reading this data on millisecond timescales).

Based on this operational requirement, this document seeks to enumerate the requirements of representing both configuration and operational state data in YANG; propose a common set of terminology; and propose a common layout for configuration and state data such that they can be retrieved from a NE. These proposals are based on the assertion that YANG models should be usable via a number of protocols (not solely IETF- defined protocols such as NETCONF and RESTCONF), and may also be used to carry data that is pushed from devices via streaming rather than polled.

2. Terminology

In order to understand the way in which a network operator or network management system may need to interact with a device, it is key to understand the different types of data that network elements may store or master:

- intended configuration - this data represents the state that the network operator intends the system to be in. This data is colloquially referred to as the 'configuration' of the system.

- applied configuration - this data represents the state that the network element is actually in, i.e., that which is currently being run by particular software modules (e.g., the BGP daemon),

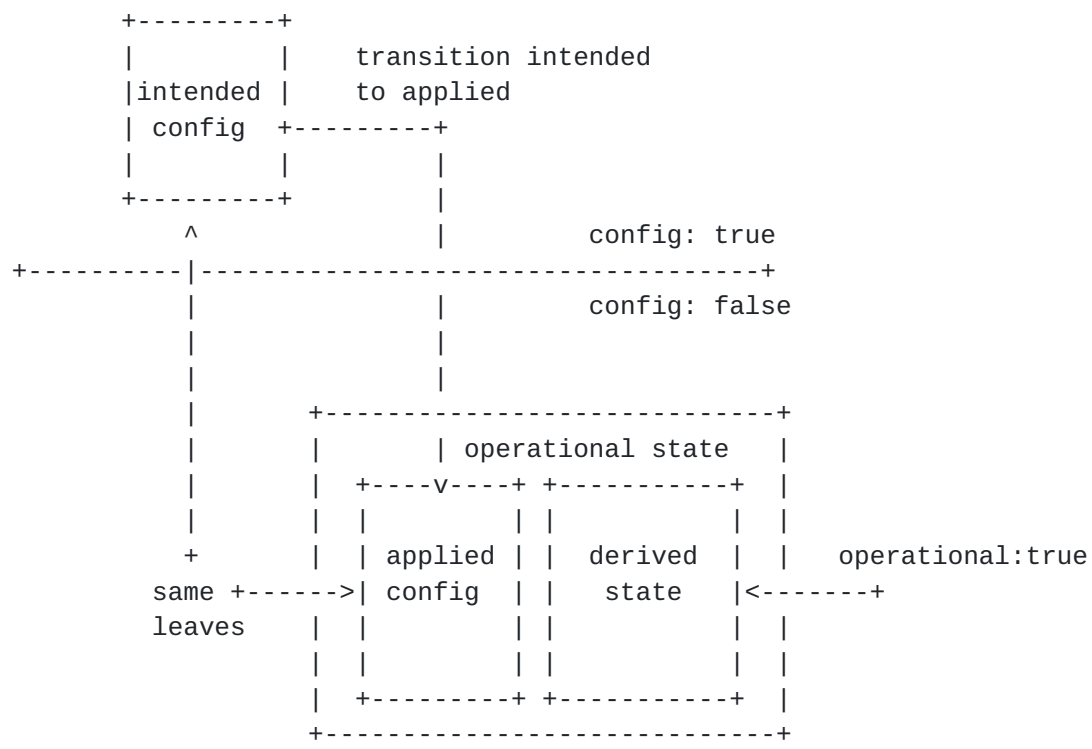
or other systems within the device (e.g., a secondary control-plane, or line card).

- derived state - this data represents information which is generated as part of the system's own interactions. For example, derived state may consist of the results of protocol interactions (the negotiated duplex state of an Ethernet link), statistics (such as message queue depth), or counters (such as packet input or output bytes).

The applied configuration and derived state can be considered as the overall 'operational' state of the NE.

When an external system desires to change the state of the network element, the changes are written to the intended configuration. This may be done directly or via a set of staged changes. The process of transitioning the intended to applied configuration may be implicit, or explicitly controlled by the network management system (NMS). Derived state is never directly influenced by the external NMS or user, since it is generated based on the systems own interactions. To this end, operational state information can be considered to be 'unknown' to the network manager.

It is notable that the intended configuration and the applied configuration represent exactly the same set of variables (leaves). These may have different values based on the current point in time (e.g., if the change has not been communicated to an external software entity), or due to missing dependencies (e.g., a particular linecard not being installed).



The relationship between intended and applied configuration, and derived state. The combination of the applied and derived state is referred to as the operational state.

Figure 1

Figure 1 shows the relationship between the different types of state referred to above. The intended configuration (which is read/write) is the only 'config: true' data. The remaining operational state (consisting of applied configuration and derived state) is read-only. Only derived state is marked as operational data.

Where the terms 'intended', 'applied', 'derived' and 'operational' are used throughout this document to refer to configuration or state, this should be read as explained above.

3. Requirement to interact with both intended and applied configuration

An operator or network management system has key requirements to be able to interact with both the intended and applied configuration. The type of interaction with each type of data does differ, however. The intended configuration is writable by the managing entity. That is, intended configuration is the means through which the NMS informs the network element of its desire to change the state of the system. An NMS may read back this intended configuration in order to

determine the state that the network element is currently trying to apply.

Once such changes have been made to the intended configuration, the NMS interacts with the read-only applied configuration to determine whether the change that was requested has been applied. The NMS can only influence changes to the applied configuration based on writing changes to the intended configuration. The applied configuration cannot be directly changed itself. It is therefore a common operation for an NMS to write to the intended configuration, and subsequently read the applied configuration to determine whether the change has been instantiated. It is therefore of great importance to have a means by which the intended and applied configuration can be easily related to one another programmatically within a single schema to avoid complex mapping between a particular intended configuration leaf and the corresponding applied configuration.

Similarly, it is also important to have operational state data for a particular entity easily related to the applied and intended configuration without requiring complex mapping. It should be noted that this does not imply that the NMS layer that is retrieving the operational state data understands the semantics of each data element, but rather that it can retrieve the required set of elements. A number of existing NMS architectures have a logical division between the elements of the system responsible for interacting with the network elements themselves, and those that are responsible for data processing, such that general data retrieval and parsing should be considered separate activities.

4. Operational requirements

The proposed modeling approach described in this document is motivated by a number of operational requirements.

4.1. Applied configuration as part of operational state

The definition of operational state in [RFC6244] includes read-only transient data that is the result of system operation or protocol interactions, and data that is typically thought of as counters or statistics. In many operational use cases it is also important to distinguish between the intended value of a configuration variable and its actual configured state, as described above. In non-transactional or asynchronous environments, for example, these may be different and it is important to know when they are different or when they have converged (see requirement #2). For this reason, we consider the applied configuration as an additional important element of the operational state. This is not considered in [RFC6244].

4.2. Support for both transactional, synchronous management systems as well as distributed, asynchronous management systems

In a synchronous system, configuration changes are transactional and committed as an atomic unit. This implies that the management system knows the success or failure of the configuration change based on the return value, and hence knows that the intended configuration matches what is on the system (i..e, what has been applied). In particular, the value of any configuration variable should always reflect the (intended) configured value. Synchronous operation is generally associated with a NETCONF-based system that provides transactional semantics for all changes.

In an asynchronous system, configuration changes to the system may not be reflected immediately, even though the change operation returns success. Rather, the change is verified by observing the state of the system, for example based on notifications, or continuously streamed values of the state. In this case, the value of a configuration variable may not reflect the intended configured value at a given point in time.

The asynchronous use case is important because synchronous operation may not always be possible. For example, in a large scale environment, the management system may not need to wait for all changes to complete if it is acceptable to proceed while some configuration values are being updated. In addition, not all devices may support transactional changes, making asynchronous operation a requirement. Moreover, using observed state to infer the configured value allows the management system to learn the time taken to complete various configuration changes.

4.3. Separation of configuration and operational state data; ability to retrieve them independently

These requirements are also mentioned in [RFC3535]:

- It is necessary to make a clear distinction between configuration data, data that describes operational state, and statistics.

- It is required to be able to fetch separately configuration data, operational state data, and statistics from devices, and to be able to compare these between devices.

4.4. Ability to retrieve operational state corresponding only to derived values, statistics, etc.

When the management system operates in synchronous mode, it should be able to retrieve only the operational state corresponding to the system determined values, such as negotiated values, protocol determined values, or statistics and counters. Since in synchronous mode the intended and applied configuration values are identical, sending the applied configuration state is redundant.

4.5. Consistent schema locations for configuration and corresponding operational state data

This requirement implies that a common convention is used throughout the schema to locate configuration and state data so that the management system can infer how to access one or the other without needing significant external context. When considering applied configuration as part of operational state (as discussed in Section 4.1), it is similarly required that the intended value vs. actual value for a particular configuration variable should be possible to locate with minimal, if any, mapping information.

This requirement becomes more evident when considering the composition of individual data models into a higher-level model for a complete device (e.g., /device[name=devXY]/protocols/routing/...) or even higher layer models maintained by network operators (e.g., /ope ratorX/global/continent[name=eur]/pop[name=paris]/device[name=devXY] /...). If each model has it's own way to separate configuration and state data, then this information must be known at potentially every subtree of the composed model.

From an operator perspective it is highly desirable that data nodes are accessible via a single data model - rather than requiring different 'views' of the same data model. This greatly simplifies NMS operation, and eliminates ambiguity for a single path. That is, it avoids the need for an NMS to provide a <RPC-call, path> tuple to uniquely identify a data node. A path should be sufficient to uniquely reference to a piece of data. Utilizing a single data model and set of paths wherever possible, ensures that this existing convention can be continued, and ambiguity of a particular path's value and meaning can be avoided.

5. Implications on modeling operational state

The requirements in Section 4 give rise to a number of new considerations for modeling operational state. Some of the key implications are summarized below.

5.1. Inclusion of applied configuration as part of operational state

This implies that a copy of the configurable (i.e., writable) values should be included as read-only variables in containers for operational state, in addition to the derived variables that are traditionally thought of as state data (counters, negotiated values, etc.).

5.2. Corresponding leaves for configuration and state

Any configuration leaf should have a corresponding state leaf. The opposite is clearly not true -- some parts of the model may only have derived state variables, for example the contents of a routing table that are populated by a dynamic routing protocols like BGP or IS-IS.

5.3. Retrieval of only the derived, or NE-generated part of the operational state

YANG and NETCONF do not currently differentiate between state that is derived by the NE, state representing statistics, and state representing applied configuration -- all state is simply marked as 'config false' or read-only. To retrieve only the state that is not part of intended configuration, we require a new way to tag such data. This is proposed in this document as a YANG extension. Alternatively, as described in [RFC6244], a new NETCONF datastore for operational state that is just for derived state could also be used to allow <get> (or similar) operations to specify just that part of the state.

5.4. Consistency and predictability in the paths where corresponding state and configuration data may be retrieved

To avoid arbitrary placement of state and configuration data containers, the most consistent options would be at the root of the model (as done in [YANG-IF]) or at the leaves, i.e., at the start or end of the paths. When operators compose models into a higher level model, the root of the model is no longer well-defined, and hence neither is the start of the path. For these reasons, we propose placing configuration and state separation at leaves of the model.

5.5. Reuse of existing NETCONF conventions where applicable

Though not a specific requirement, models for operational state should take advantage of existing protocol mechanisms where possible, e.g., to retrieve configuration and state data. As mentioned above, this does not mean that the solution for modeling operational state and configuration data should be limited to NETCONF architecture or protocols.

6. Proposed operational state structure

Below we show an example model structure that meets the requirements described above for all three types of data we are considering:

- intended configuration
- applied configuration
- derived state

6.1. Example model structure

The example below shows a partial model (in ascii tree format) for managing Ethernet aggregate interfaces (leveraging data definitions from [RFC7223]):

```
   +--rw interfaces
       +--rw interface* [name]
          +--rw name       -> ../config/name
          +--rw config
          |    ...
          +--ro state
          |  | ...
          |  +--ro counters
          |     +--ro discontinuity-time    yang:date-and-time
          |     +--ro in-octets?            yang:counter64
          |     +--ro in-unicast-pkts?      yang:counter64
          |     +--ro in-broadcast-pkts?    yang:counter64
          |     +--ro in-multicast-pkts?    yang:counter64
          |     +--ro in-discards?          yang:counter64
          |     +--ro in-errors?            yang:counter64
          |     +--ro in-unknown-protos?    yang:counter64
          |     +--ro out-octets?           yang:counter64
          |     +--ro out-unicast-pkts?     yang:counter64
          |     +--ro out-broadcast-pkts?   yang:counter64
          |     +--ro out-multicast-pkts?   yang:counter64
          |     +--ro out-discards?         yang:counter64
          |     +--ro out-errors?           yang:counter64
          +--rw aggregation!
             +--rw config
             |  +--rw lag-type?    aggregation-type
             |  +--rw min-links?   uint16
             +--ro state
             |  +--ro lag-type?    aggregation-type
             |  +--ro min-links?   uint16
             |  +--ro members*     ocif:interface-ref
             +--rw lacp!
                +--rw config
                |  +--rw interval?   lacp-period-type
                +--rw members* [interface]
                |  +--rw interface    ocif:interface-ref
                |  +--ro state
                |     +--ro activity?          lacp-activity-type
                |     +--ro timeout?           lacp-timeout-type
                |     +--ro synchronization?   lacp-synch-type
                |     +--ro aggregatable?      boolean
                |     +--ro collecting?        boolean
                |     +--ro distributing?      boolean
                +--ro state
                   +--ro interval?   lacp-period-type
```

In this model, the path to the intended configuration (rw) items at the aggregate interface level is:

/interfaces/interface[name=ifName]/aggregation/config/...

The corresponding applied configuration and derived state is located at:

/interfaces/interface[name=ifName]/aggregation/state/...

This container holds a read-only copy of the intended configuration variables (lag-type and min-links) - the applied configuration - as well as a generated list of member interfaces (the members leaf-list) for the aggregate that is active when the lag-type indicates a statically configured aggregate (which is derived state). Note that although the paths to config and state containers are symmetric, the state container contains additional derived variables.

The model has an additional hierarchy level for aggregate interfaces that are maintained using LACP. For these, the configuration path is:

/interfaces/interface[name=ifName]/aggregation/lacp/config/...

with the corresponding state container (in this case with only the state corresponding to the applied configuration) at:

/interfaces/interface[name=ifName]/aggregation/lacp/state/...

There is an additional list of members for LACP-managed aggregates with only a state container:

/interfaces/interface[name=ifName]/aggregation/lacp/
members[name=ifName]/state/...

Note that it is not required that both a state and a config container be present at every leaf. It may be convenient to include an empty config container to make it more explicit to the management system that there are no configuration variables at this location in the data tree.

Finally, we can see that the generic interface object also has config and state containers (these are abbreviated for clarity). The state container has a subcontainer for operational state corresponding to counters and statistics that are valid for any interface type:

/interfaces/interface[name=ifName]/state/counters/...

7. Discussion and observations

A number of issues have been raised with the proposed solution, which are documented below, along with the authors observations relating to these issues.

1. The proposed solution decreases the readability of a YANG data model for some, or the ease of writing a model for others. It is difficult to make this judgment without being subjective - the complexity in model writing (as is noted in the above section) is only at the expense of meeting the operational requirement described in this document. The authors consider that this is a fair trade-off between one-time modeling complexity. It could also be observed that a common convention for representing operational state data alongside configuration improved readability.

2. Data is duplicated on the wire by this proposal. The intention of defining a set of annotations for data (operational: true, or the data-type flag proposed below) is in order to allow RPCs to be defined which return only specific types of data. For example, a >get-operational< call may return only values with operational: true so that an NMS can return a specific set of data to the requesting entity.

3. The proposal does not allow items that are not configured, configured but not present, or system configured. A common example which is quoted is where there are elements that are not configured, or are system-generated based on some other configuration. For example, consider a model whereby an 'all' interface is configured, which corresponds to all interfaces on the system. In this case, the intended configuration should include only the 'all' interface which is configured. This intended configuration should be reflected to the applied configuration. The operational state should contain per-interface (e.g., eth0, Fa0/1) values relating to the interface entities that exist in the network. The intended configuration corresponds solely to a particular interface (e.g., eth0) -- there should be no corresponding 'intended' configuration. In these cases, there is no 'intended' configuration for an entity, but there is an 'applied' configuration present. One challenge here relates to the fact that YANG's list semantics currently imply that that the "config true" interface-name leaf has been set - in practice, it is unlikely that this list key is actually configurable in any real system (it must correspond to a real interface, which has an explicit name according to the system implementation). Additionally, this could be resolved with the alternative map type described later in this document.

4. It is not clear what to do when the intended and applied configuration differ. The proposal made in this document makes no presumption as to the actions that are taken when intended and applied leaves for a certain value differ. In fact, it is the expectation of the authors that there is separation between elements of the NMS that are responsible for retrieving data from network elements, as opposed to those that need to understand process this data. The fact that this layer interacting with the network can retrieve both intended and applied configuration, and find the corresponding operational state data in a consistent manner is independently useful regardless of whether the semantics of the contained data are understood.

5. An operational-path statement could be used to point between intended and applied configuration. Essentially, this proposal moves the mapping dictionary on a per-leaf basis within the data model itself. It appears to be a more complex solution that the proposed approach within this document which does not require any need to build a per-leaf mapping.

6. Models that do not follow the proposed pattern would not be usable. Models that do not follow the structural convention for modeling operational state data would require some refactoring to meet the requirements described in this document. However, by following the design pattern for YANG grouping described in Section Section 8.1.1 it becomes possible to leverage existing modules by importing them and reusing the groupings. More specifically, if models are designed with only configuration or state related data leaf nodes in groupings, another model could create the required structure and reuse these groupings.

8. Impact on model authoring

One drawback of structuring operational and configuration data in this way is the added complexity in authoring the models, relative to the way some models are currently built with state and config split at the root of the individual model (e.g., in [RFC7223], [RFC7317], and [IETF-RTG]). Moving the config and state containers to each leaf adds a one-time modeling effort, which is somewhat dependent on the model structure itself (how many layers of container hierarchy, number of lists, etc.) However, we feel this effort is justified by the resulting simplicity with which management systems can access and correlate state and configuration data.

8.1. Modeling design patterns

We propose some specific YANG modeling design patterns that are be useful for building models following these conventions.

8.1.1. Basic structure

Since leaves that are created under the 'config' container also appear under the 'state' container, it is recommended that the following conventions are used to ensure that the schema remain as simple as possible:

- A grouping for the intended configuration data items is created - with a specific naming convention to indicate that such variables are configurable, such as a suffix like '-config' or '_config'. For example, the OpenConfig BGP model [OC-BGP] adopts the convention of appending "_config" to the name of the grouping.

- A grouping for the derived state data items is created, with a similar naming convention as above, i.e., with a suffix such as '-state' or '_state'. The BGP model uses "_state".

- A 'structural' grouping is created that instantiates both the 'config' and 'state' containers. The 'config' container should include the "-config" grouping, whilst the state container has both the "-config" and "-state" groupings, along with the 'config false' statement.

A simple example in YANG is shown in Appendix B.

8.1.2. Handling lists

In YANG 1.0, lists have requirements that complicate the creation of the parallel configuration and state data structures. First, keys must be children of the list; they cannot be further down the data hierarchy within a subsequent container. For example, the 'interface' list cannot be keyed by /interfaces/interface/config/name. Second, YANG requires that the list key is part of the configuration or state data in each list member.

We consider two possible approaches for lists:

1. list keys appear only at the top level of the list, i.e., not duplicated under the 'config' or 'state' containers within the list

2. the data represented by the list key appears in the config and state containers, and a key with type leafref is used in the top

level of the list pointing to the corresponding data node in the config (or state) container.

Option 1 has the advantage of not duplicating data, but treats the data item (or items) that are keys as special cases, i.e., not included in the config or state containers. Option 2 is appealing in that configurable data always appears in the config container, but requires an arguably unnecessary key pointing to the data from the top level of the list.

Appendix C shows a simple example of both options.

8.1.3. Selective use of state data from common groupings

In a number of cases, it is desirable that the same grouping be used within different places in a model - but state information is only relevant in one of these paths. For example, considering BGP, peer configuration is relevant to both a "neighbor" (i.e., an individual BGP peer), and also to a peer-group (a set of peers). Counters relating to the number of received prefixes, or queued messages, are relevant only within the 'state' container of the peer (rather than the peer-group). In this case, use of the 'augment' statement to add specific leaves to only one area of the tree is recommended, since it allows a common grouping to be utilized otherwise.

8.1.4. Non-corresponding configuration and state data

There are some instances where only an operational state container is relevant without a corresponding configuration data container. For example, the list of currently active member interfaces in a LACP-managed LAG is typically reported by the system as operational state that is governed by the LACP protocol. Such data is not directly configured. Similarly, counters and statistics do not have corresponding configuration. In these cases, we can either omit the config container from such leaves, or provide an empty container as described earlier. With both options, the management system is able to infer that such data is not configurable.

9. YANG language considerations

In adopting the approach described in this document for modeling operational state data in YANG, we encounter several language limitations that are described below. We discuss some initial thoughts on possible changes to the language to more easily enable the proposed model for operational state modeling.

9.1. Distinguishing derived operational state data and applied configuration

As mentioned in Section 4, we require a way to separately query operational state that is not part of applied configuration (e.g., protocol-determined data, counters, etc.). YANG and NETCONF do not distinguish types of operational state data, however. To overcome this, we currently use a YANG language extension to mark such data as 'operational: true'. Ideally, this could be generalized beyond the current 'config: true / false' to mark "data-type: intended", "data-type: applied", "data-type: derived" to allow filtering of particular types of data by a protocol RPC.

9.2. YANG lists as maps

YANG has two list constructs, the 'leaf-list' which is similar to a list of scalars (arrays) in other programming languages, and the 'list' which allows a keyed list of complex structures, where the key is also part of the data values. As described in Section 8.1.2, the current requirements on YANG list keys require either duplication of data, or treating some data (i.e., those that comprise list keys) as a special case. One solution is to generalize lists to be more like map data structures found in most modern programming languages, where each list member has a key that is not required to part of the configuration or state data, and also not subject to existing "config-under-state limitations. This allows list keys to be arbitrarily defined by the user if desired, or based on values of data nodes. In the latter case, the specification of which data nodes are used in constructing the list key could be indicated in the meta-data associated with the key.

9.3. Configuration and state data hierarchy

YANG does not allow read-write configuration data to be child nodes of read-only operational state data. This requires the definition of separate state and config containers as described above. However, it may be desirable to simplify the schema by 'flattening', e.g., having the operational state as the root of the data tree, with only config containers needed to specify the variables that are writable (in general, the configuration data is much smaller than operational state data). Naming the containers explicitly according the config / state convention makes the intent of the data clear, and should allow relaxing of the current YANG restrictions. That is, a read-write config container makes explicit the nature of the enclosed data even if the parent data nodes are read-only. This of course requires that all data in a config container are in fact configurable -- this is one of the motivations of pushing such containers as far down in the schema hierarchy as possible.

10. Security Considerations

This document addresses the structure of configuration and operational state data, both of which should be considered sensitive from a security standpoint. Any data models that follow the proposed structuring must be carefully evaluated to determine its security risks. In general, access to both configuration (write) and operational state (read) data must be controlled through appropriate access control and authorization mechanisms.

11. References

11.1. Normative references

[RFC6020] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[RFC6244] Shafer, P., "An Architecture for Network Management Using NETCONF and YANG", RFC 6244, June 2011.

[RFC3535] Schoenwaelder, J., "Overview of the 2002 IAB Network Management Workshop", RFC 3535, May 2003.

[RFC7223] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 7223, May 2014.

[RFC7317] Bierman, A. and M. Bjorklund, "A YANG Data Model for System Management", RFC 7317, August 2014.

11.2. Informative references

[IETF-RTG] Lhotka, L., "A YANG Data Model for Routing Management", draft-ietf-netmod-routing-cfg-16 (work in progress), October 2014.

[OC-BGP] Shaikh, A., D'Souza, K., Bansal, D., and R. Shakir, "BGP Configuration Model for Service Provider Networks", draft-shaikh-idr-bgp-model-01 (work in progress), March 2015.

Appendix A. Acknowledgments

The authors are grateful for valuable input to this document from: Lou Berger, Martin Bjorklund, Paul Borman, Chris Chase, Raymond Cheh, Feihong Chen, Benoit Claise, Josh George, Carl Moberg, Jason Sterne, Jim Uttaro, and Kent Watsen.

Appendix B. Example YANG base structure

Below we show an example of the basic YANG building block for organizing configuration and operational state data as described in Section 6

```
grouping example-config {
    description "configuration data for example container";

    leaf conf-1 {
      type empty;
    }

    leaf conf-2 {
      type string;
    }
  }

  grouping example-state {
    description
      "operational state data (derived, counters, etc.) for example
      container";

    leaf state-1 {
      type boolean;
      operational true;
    }

    leaf state-2 {
      type string;
    }

    container counters {
      description
        "operational state counters for example container";

      operational true;

      leaf counter-1 {
        type uint32;
      }

      leaf counter-2 {
        type uint64;
      }
    }
  }
```

```
    grouping example-structure {
      description
        "top level grouping for the example container -- this is used
        to put the config and state subtrees in the appropriate
        location";

      container example {
        description
          "top-level container for the example data";

        container config {

          uses example-config;

        }

        container state {

          config false;
          uses example-config;
          uses example-state;
        }
      }
    }

    uses example-structure;
```

The corresponding YANG data tree is:

```
   +--rw example
      +--rw config
      |  +--rw conf-1?   empty
      |  +--rw conf-2?   string
      +--ro state
         +--ro conf-1?     empty
         +--ro conf-2?     string
         +--ro state-1?    boolean
         +--ro state-2?    string
         +--ro counters
            +--ro counter-1?   uint32
            +--ro counter-2?   uint64
```

Appendix C. Example YANG list structure

As described in Section 8.1.2, there are two options we consider for building lists according to the proposed structure. Both are shown in the example YANG snippet below. The groupings defined above in Appendix B are reused here.

```
   grouping example-no-conf2-config {
       description
       "configuration data for example container but without the conf-2
       data leaf which is used as a list key";

       leaf conf-1 {
         type empty;
       }

     }

     grouping example-structure {
       description
         "top level grouping for the example container -- this is used
         to put the config and state subtrees in the appropriate
         location";

       list example {

         key conf-2;
         description
           "top-level list for the example data";

         leaf conf-2 {
           type leafref {
             path "../config/conf-2";
           }
         }

         container config {

           uses example-config;

         }

         container state {

           config false;
           uses example-config;
           uses example-state;
         }
       }

       list example2 {

         key conf-2;
         description
           "top-level list for the example data";
```

```
        leaf conf-2 {
          type string;
        }

        container config {

          uses example-no-conf2-config;

        }

        container state {

          config false;
          uses example-no-conf2-config;
          uses example-state;
        }
      }
    }

    uses example-structure;
```

The corresponding YANG data tree is shown below for both styles of lists.

```
+--rw example* [conf-2]
   |  +--rw conf-2    -> ../config/conf-2
   |  +--rw config
   |  |  +--rw conf-1?   empty
   |  |  +--rw conf-2?   string
   |  +--ro state
   |     +--ro conf-1?     empty
   |     +--ro conf-2?     string
   |     +--ro state-1?    boolean
   |     +--ro state-2?    string
   |     +--ro counters
   |        +--ro counter-1?   uint32
   |        +--ro counter-2?   uint64
   +--rw example2* [conf-2]
      +--rw conf-2    string
      +--rw config
      |  +--rw conf-1?   empty
      +--ro state
         +--ro conf-1?     empty
         +--ro state-1?    boolean
         +--ro state-2?    string
         +--ro counters
            +--ro counter-1?   uint32
            +--ro counter-2?   uint64
```

Appendix D. Changes between revisions -00 and -01

The -01 revision of this documents reflects a number of discussions with implementors and members of several IETF working groups, including NETMOD. Major changes from the prior version are summarized below.

- Updated introduction to provide additional background on operational requirements.
- Added a detailed terminology section and diagram to provide definitions of different types of modeled data based on working group discussions.
- Added new discussion section summarizing issues that have been raised with the proposal as well as operator observations and comment.

Authors' Addresses

Rob Shakir
BT
pp. C3L, BT Centre
81, Newgate Street
London EC1A 7AJ
UK

Email: rob.shakir@bt.com
URI: http://www.bt.com/

Anees Shaikh
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
US

Email: aashaikh@google.com

Marcus Hines
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
US

Email: hines@google.com